Static Analysis of Java for Distributed and Parallel Programming

Isabelle Attali, Denis Caromel, Romain Guider

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Static Analysis of Java for distributed and parallel programming

Isabelle Attali — Denis Caromel — Romain Guider

N° 3634

Mars 1999

THÈME 2

ISSN 0249-6399

Static Analysis of Java for distributed and parallel programming

Isabelle Attali , Denis Caromel , Romain Guider

Thème 2 — Génie logiciel
et calcul symbolique
Projet Oasis

Rapport de recherche n° 3634 — Mars 1999 — 33 pages

Abstract: We investigate the use of static analysis for building distributed and parallel programs in Java.

We first quickly explain a principle of seamless sequential, multithreaded and distributed programming using Java, in order to enhance code reuse and code evolution. We exhibit conditions on the graph of objects to detect activable objects and transform a sequential program into a distributed or parallel program using active objects.

We then present a static analysis based on an abstract interpretation of a Java subset, which provides approximate sets of activable objects.

Finally, we illustrate our algorithm and results with one example.

Key-words: Static Analysis, Abstract Interpretation, Operational Semantics, Parallel and distributed Object-Oriented Programming, Java.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 —Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Analyse statique de programmes Java pour la programmation parallèle et répartie

Résumé : Nous étudions l'utilisation de l'analyse statique pour la construction d'applications Java parallèles et réparties.

Nous expliquons tout d'abord le principe et le modèle pour une programmation Java parallèle ou répartie, de manière à favoriser la réutilisation de code séquentiel et faciliter la maintenance et l'évolution. Nous exhibons des conditions sur le graphe d'objets pour détecter des objets activables et transformer un programme séquentiel en un programme parallèle ou réparti utilisant des objets actifs.

Nous présentons ensuite une analyse statique fondée sur une interprétation abstraite d'un sous-ensemble du langage Java qui permet de définir des sous-ensembles approximés d'objets activables.

Enfin, nous illustrons notre algorithme et nos résultats par un exemple.

Mots-clés : Analyse statique, Interprétation Abstraite, Sémantique opérationnelle, Programmation à objets parallèle et répartie

1 Introduction

In this article, we investigate the use of static analysis for building distributed and parallel programs in Java.

One major issue of object-oriented programming is reusability through inheritance, polymorphism, and dynamic binding. This feature has been also studied and enhanced in the context of parallelization and distribution. Several concurrent object-oriented programming languages have been designed, see for instance Hybrid [32], Pool [3], ABCL [43], DROL [42], and more recently Java [23]. Also, many object-oriented languages have been extended to address concurrency, parallelism, and distribution issues, see for instance ConcurrentSmalltalk [44], Distributed Smalltalk [8], Eiffel// [10], Java// [13].

Most of these works offer a setting for easy parallel, distributed, and concurrent programming, starting from a sequential application, and making it run on a parallel or distributed architecture. Some languages provide a unified syntax (no syntactic extension) for both sequential and parallel versions. This feature is critical for reuse and is also beneficial for the proof since the actions modeling the program are identical in both cases. Another problem indeed is to be able to guarantee that the semantics of the original sequential version is preserved in a parallel and distributed setting. This problem has been tackled in various works, using different modelizations (π-calculus, operational semantics, TLA (Temporal Logic of Actions)) for instance in [30,28,5,6], these work are to be generalized and extended to the detection of how to parallelize.

We propose to analyse a sequential program, and to exploit the results of this analysis with the intervention of the (end-)user, in order to provide a parallel version of the original program which has two properties: (1) take advantage of reuse in order to impose as few modifications as possible, (2) guarantee the preservation of sequential semantics. The distributed programs are using primitives of Java// (Java Parallel [13]), a Java library for distributed, concurrent, and parallel computing.

Static analysis has been widely used in compile-time optimizations of various programming languages [39,26,38], (static) debugging tools [21], and more recently have been applied to optimization of object-oriented languages [35,20]. Also related to object-oriented programming, static analysis has been used as a checking mechanism for a type-system [2] including constraint on sharing.

We propose an analysis in the family of static analysis of dynamic (during run-time) program properties. More precisely, our analysis, based on Abstract Interpretation [17] provides program transformation for parallel or distributing programming.

Formally, abstract interpretation relies on the notion of discrete approximation, that is, replacing the reasoning on a concrete exact semantics by a computation on an abstract approximate semantics. In this paper, we define an operational semantics of a large subset of Java (excluding threads, class variables and methods, and dynamic class loading). From this (concrete) semantics, we define an abstract approximate semantics, made up with abstract inference rules and abstractions of the semantic structures. We formally define the notion of *accessibility* of objects in terms of the object graph topology. From this notion, we express the property of *activability* of objects, avoiding interferences between computations. Our static

analysis makes it possible to statically determine classes of objects where this property is verified and as a consequence, what objects can be transformed into active objects.

The next section of this paper presents the underlying model of distribution, describes a concrete dynamic semantics of a subset of Java, and gives sufficient conditions, based on this semantics for activable objects. Section 3 focuses on the static analysis with the description of abstractions for two components used in the operational semantics: call-stacks and object-graphs. We deal both with intra-procedural and inter-procedural transitions. Section 4 presents some case studies illustrating the power of the analysis in the transformation of a sequential program into a parallel or distribued program. We illustrate as often as possible the concepts and techniques used this article with well-known examples (binary tree, linked lists, sieve). Section 5 is a discussion of related work. Finally, Section 6 (Conclusion) briefly discusses our contribution and outlines future work.

2 Distribution of Object-Oriented Programs

Our goal in this section is twofold:

- briefly introduce and explain the active object model we intend to use for distribution and parallelism (the Java// model and library),
- formally define a sufficient condition on an object that guarantees it can be safely activated.

First, we give an overview of the Java// model and library, and explain how one can use it as a target system for distributing applications. We pinpoint some of the crucial features that make it possible, with very limited changes, to turn standard objects into active ones to be distributed. A small example illustrates that technique (a binary search tree).

Then, we give a sight at an operational semantics that describes execution of Java programs. The next subsection defines, based on that semantics, the set of objects possibly affected (the *accessibility* of an object) by method invocations on a given object. We rely on the widely accepted conditions for parallelization: two operations can be conducted concurrently if they access (read or write) distinct sets of objects. We exhibit a sufficient condition on accessibilities (equivalently on objects) that ensures, for a given object o, that methods invoked on o do not interfere with method invocation on objects that are not in the accessibility of o. Finally, we show that if such objects are activated, then the graph of active objects is a tree. The consequence being that operations on a given object in the parallel program are triggered in the same order as they are on the corresponding object in the sequential program.

2.1 Model of Distribution

The target model of distribution that we use has been studied and improved along several experiments, both practical and more formal [11,12,5,4]. The current experimentation and

implementation are done within Java with a library named Java// [13]. Its main goal is to improve simplicity and reuse in the programming of distributed object systems.
The Java// model uses by default the following principles and features:

- heterogeneous model with both passive and active objects (processes, actors);
- sequential processes;
- unified syntax between message passing and inter-process communication;
- systematic asynchronous communications towards active objects;
- wait-by-necessity (automatic and transparent futures);
- automatic continuations (a transparent delegation mechanism);
- no shared passive objects (call-by-value between processes);
- centralized and explicit control by default;
- polymorphism between standard objects (threads) and remote objects.

Based on an heterogeneous model, and thanks to the absence of sharing, a system is always structured as several *sub-systems*. Each subsystem is defined as an active root object, and all the standard objects (not active) that it can reach.

Given a sequential Java program, it takes only minor modifications from the programmer in order to turn it into a multithreaded, parallel, or distributed one. Java// actually only requires instantiation code to be modified in order to transform a standard object into an active one. Here is a sample of code with several techniques for turning a passive instance of class A into an active, possibly remote, one. A standard object created through such a statement:

```
A a = new A ("foo", 7) ;
```

can become either:

- instantiation-based:
```
Object[] params={"foo", 7};
A a =(A) Javall.newActive ("A",params, Node);
```
- class-based:
```
class pA extends A implements Active {}
Object[] params={"foo", 7};
A a =(A) Javall.newActive ("pA", params, Node);
```
- object-based:
```
A a = new A ("foo", 7) ; // No change
a = (A) Javall.turnActive (a, Node);
```

All these techniques create an active object, an instance of class `A` or `pA` on a given node. The active object just created its own thread that executes methods invoked on this object in a default FIFO order. The semantics of calls to such an object are transparently asynchronous, with no code modification being required on the caller's side.

Instantiation-based creation is much of a convenience technique. It allows the programmer to create an active instance of `A` with a FIFO behavior without defining any new class. In the class-based creation, given a class `A`, the programmer writes a subclass `pA` that inherits directly from `A` and implements the specific marker interface `Active`. He or she may

```
package fr.inria.sloop.javall.examples.binarytree;
public class BinaryTree extends Object
{
    protected int key;      // Key for accessing the value contained in this node
    protected Object value; // Actual value contained in this node
    protected BinaryTree leftTree; // The two subtrees
    protected BinaryTree rightTree;

    public BinaryTree ()
    {
        this.value = null;     // On creation, the node does not contain a value
        this.leftTree = null;  // Nor does it have any child
        this.rightTree = null;
    }

    //   Inserts a (key, value) pair in the subtree that has this node as its root
    public void put (int key, Object value)
    {
        // This node is empty, let's use it
        if ((this.leftTree==null) && (this.rightTree==null))) // Is leaf
        {
            this.key = key; this.value = value;
            this.createChildren ();
            return;
        }
        // Replaces the current value with a new one
        else if (key==this.key)
        {
            this.value = value;
        }
        // smaller keys are on the left,
        else if (key<this.key)
        {
            this.leftTree.put (key, value);
        }
        // greater keys on the right,
        else
        {
            this.rightTree.put (key, value);
        }
        return;
    }

    //  Retrieve a value from a key in the subtree that has this node as its root
    public Object get (int key)
    {
        // We have reached a leaf of the tree and no key matching the parameter 'key'
        // has been found. This is a miss.
        if ((this.leftTree==null) && (this.rightTree==null))) // Is leaf
        {
            return null;
        }
        // We have found the node that contains the value we're looking for
               else if (key==this.key)
        {
            return this.value;
        }
        // The current key is greater than the search key, let's continue on the left
        else if (key<this.key)
        {
            return this.leftTree.get (key);
        }
        // The current key is smaller than the search key, let's continue on the right
        else
        {
            return this.rightTree.get (key);
        }
    }

    // Creates two empty leaves as children
    protected void createChildren ()
    {
        this.leftTree = new BinaryTree (); this.rightTree = new BinaryTree ();
        return;
    }
}
```

Fig. 1. Sequential Binary Tree

```
package fr.inria.sloop.javall.examples.binarytree;
import fr.inria.sloop.javall.*;

public class ActiveBinaryTree extends BinaryTree implements javall.Active
{   /*  This of course is a bulky implementation, one could provide
        a more subtle scheme, such as creating active objects only
        for a given depth n of the tree (so that there cannot
        be more that 2**n active objects)  */
   protected void createChildren ()
    {   this.leftTree = (BinaryTree) Javall.newActive ("ActiveBinaryTree", null, null);
        this.rightTree = (BinaryTree) Javall.newActive ("ActiveBinaryTree", null, null);
        return;
}}
```

Fig. 2. Active Binary Tree

also provide a live method in class `pA` for giving a specific activity or managing synchronization. The object-based technique enables a programmer to attach an active behavior to an existing object at any time after its creation. This is especially useful when one does not have access to the code that creates the standard object to be made active, however, this technique is not used for the distribution transformations described in the current paper.

Once the active object is created, it automatically features the principles described previously. Among them, a few are critical for the goal of this paper: polymorphism between objects and active objects (allows the transformations), sequential processes without shared objects (no interleaving), asynchrony of calls and automatic continuations (avoids deadlocks and allows parallelism), wait-by-necessity (automatically respects data dependencies).

We have been studying various case studies of parallelizations, and it appears that, with these features, if the graph of objects at execution is "a tree", then we can safely turn the objects of the graph into active ones. This property have been formally studied within various frameworks [4,5,6] and demonstrated on either examples or subsets of the model, and we are currently working on its generalization. The goal of this paper is not to formally prove that property, but rather to exploit it in order to detect the places in a system where we can apply it for the sake of distribution or parallelization. Being a property on dynamic structures, the graph of objects and its topology at execution, it requires static analysis to uncover the places where it holds.

Figures 1, 2, 3 present an example of parallelization: a binary search tree. Applied on this example, the goal that we pursue in this paper is to help the programmer to identify the places where the `BinaryTree` class can be transformed into an active object while the semantics remains constant. In Figure 1, the system will point out that the instantiation of `BinaryTree`:

```
myTree = new BinaryTree (); // Instantiating a standard version
```

can be replace with an active binary tree. So the user, possibly with the help of semi-automatic tools, will replace the line above with:

```
myTree=(BinaryTree) Javall.newActive ("ActiveBinaryTree",null,null);
```

and write the class `ActiveBinaryTree`.

```
package fr.inria.sloop.javall.examples.binarytree;
import fr.inria.sloop.javall.*;

public class TestBT
{   public static void main (String[] args)
    {   BinaryTree myTree;
        myTree = new BinaryTree (); // Instantiating a standard version
              // To get an active version, just comment the line above,
              // and comment out the line below
        // myTree = (BinaryTree) Javall.newActive ("ActiveBinaryTree", null, null);
        // * First parameter:  get an active instance of class ActiveBinaryTree
        // * Second ('null'):  instantiates with empty (no-arg) constructor
        //                     'null' is a convenience for 'new Object [0]'
        // * Last ('null'):    instantiates this object on the current host,
        //                     within the current virtual machine
             // Use either of the two versions: standard or active
             // through polymorphism
        TestBT.useBinaryTree (myTree);
        return;
    }
// Note that this code is the same for the passive or active version of the tree
    protected static void useBinaryTree (BinaryTree bt)
    {   String s1; String s2;
        bt.put (1, "one");     // We insert 4 elements in the tree, non-blocking
        bt.put (2, "two");
        bt.put (3, "three");
        bt.put (4, "four");
           // Now we get all these 4 elements out of the tree
           // method get in class BinaryTree returns a future object if
           // bt is an active object, but as System.out actually calls toString()
           // on the future, the execution of each of the following 4 calls
           // to System.out blocks until the future object is available.
        System.out.println ("Value associated to key 2 is "+bt.get (2));
        System.out.println ("Value associated to key 1 is "+bt.get (1));
        System.out.println ("Value associated to key 3 is "+bt.get (3));
        System.out.println ("Value associated to key 4 is "+bt.get (4));
           // When using variables, all the instructions are non-blocking
        bt.put (2, "twoBis");
        s2 = bt.get (2); // non-blocking
        bt.put (2, "twoTer");
        s2 = bt.get (2); // non-blocking
        s1 = bt.get (1); // non-blocking
           // Blocking operations
        System.out.println ("Value associated to key 2 is "+ s2 ); // prints "twoTer"
        System.out.println ("Value associated to key 1 is "+ s1 ); // prints "one"
        return;

}}
```

Fig. 3. Binary Tree: example of main program

The next four subsections are dedicated to the formal definition of a Java semantics, that is needed to later on define the static analysis, and then formally define a necessary condition for objects to be activable.

2.2 A Java Semantics

Java subset:
The main features that we do not treat are:

- threads (if the programmer intended to parallelize his program using Java//, he will probably not intend to use Java threads at the same time),
- class variables and class methods: static memory allocation via the keyword `static` can be easily taken into account when one can handle dynamic structures allocated on the heap,
- dynamic class loading: can not be easily included because some programs may actually generate classes and load them. This is is particularly difficult to represent using abstract interpretation.

exp ::=	id	$inst$ ::=	$return(exp)$
	$\mid access(exp, id)$		$\mid while(exp, inst)$
	$\mid call(exp, exps)$		$\mid if(exp, inst, inst')$
	$\mid new(id, exps)$		$\mid block$
	$\mid assign(exp, exp)$		$\mid exp$
	$\mid none$		$\mid push$
			$\mid eval(exp)$
$exps$ ::=	$exp^\star$		$\mid exec(inst)$
$block$ ::=	$inst^\star$		$\mid wait(exp)$

Fig. 4. Abstract Syntax

We also restricted the number of statements treated to a subset that can be easily completed: `if`, `while`, `try-catch-finally`, `return`, `break`, `continue`, `label`, `throw`. We only consider here `while` and `if-else` statements for simplicity (see Figure 4 for a definition of the abstract syntax). From the class definition, we only retain sub-classing and method overriding. We omit instance-variable overriding and method overloading because those feature can be resolved statically (e.g. during type-checking) and as such do not introduce any semantic difficulty. From the base types we retain only `int` and `boolean`.

Semantics specification style Different formal semantics specifications of Java programs have been published, but none can be easily used to design a static analysis by abstract interpretation. Nipkow et al. [33] give a natural semantics specification of the dynamic semantics of Java. The problem with natural semantics is that it only specifies the behavior of programs that terminate and we need to represent both terminating and non-terminating programs. Attali et al. [7] have designed a structural operational semantics. It represents current program points by replacing call statements with a closure built from the actual method body that is called. In the presence of a recursive method, we would have to design a complicated abstraction of program points because trees that represent program points would not be in finite number.

Cousot and Cousot [18], and Schmidt [40,41] have proposed solutions to overcome these problems, but we preferred concentrating on the problem of heap abstraction by using a semantics formalism for which abstract interpretation is well understood: transition systems.

Semantics Domains The domains we use to represent the states of our interpreter are the following:

$$\begin{aligned} Val &= Ref \cup Int \cup Bool \\ Env &= Id \rightarrow Val \\ Obj &= Ref \times Type \times Env \\ Objs &= \mathcal{P}(Obj) \\ Act &= Inst \times Id \times Type \times Env \\ Stack &= Act^\star \\ State &= Stack \times Objs \times Val \end{aligned}$$

A state consists of a stack of method activations, an object graph, and a place-holder for values. Method activations (Act) record information about a currently executed method-call : the instruction that is to be executed, the name of the method, the name of the class where it has been defined, and a local environment. The place-holder for values is used to store the value returned by a terminated method execution.

Environments are mappings from identifiers to values. They include a special pair $\langle this, o_{this} \rangle$ where o is the object target of the method-call. The interpretation of a program starts by pushing an activation corresponding to the method `main` of the program onto the call-stack.

The domain Ref provides a potentially infinite reserve of distinct elements (integers for example) that are used to identify objects (references). Object graphs are sets of objects. Objects are structures that record their reference, the class that instantiated them, and an environment that maps their attributes names to their values.

The transition relation between states The execution of a program is the trace of a transition relation defined over $State \times State$. We briefly describe it.

Normalization of programs The semantics actually interprets programs that are in a normalized form. This form is used by Sagiv, Reps, and Wilhelm [38]. Assignments are restricted to the forms:

- `x = y` //assigns the value of a local variable to another one
- `x = y.s` //stores an object attribute into a local variable
- `x.s = y` //assigns an object attribute the value of a local variable
- `x = new ...`
- `x = call( ... )`

Each instruction of the form presented above have to be preceded by an instruction that nullifies its left-hand side. In addition, arguments of method-calls are restricted to be local variables (this holds for constructor invocations as well) and any access to an attribute of the current object (referenced by `this`) in the current method is prefixed by `this` so that access to local variables and to attributes can be distinguished syntactically.

This normalization is simply realised by introducing auxiliary local variables to store partial results of expression evaluation. The size of normalized programs is linearly bounded by the size of the initial program. Normalisation is used to simplify presentation. In our current implementation, operand stacks are used to evaluate expressions, and positions in this stack serve as names in the abstract interpretation.

Method-call At first, the argument list is evaluated, then the target of the call is fetched from the local environment (the target expression is restricted to be a local variable). Then method lookup gets the right method body. A new activation is build where parameters are bound to their respective values in an environment. Finally, that activation is pushed onto the call-stack. In the waiting activation, the `call` instruction is replaced by a `wait` instruction.

Returning from a method The interpretation of the instruction `return` gets the expression returned (when there is one) from the local environment and assigns its value to the place-holder. Then the terminated activation is popped. The interpretation goes on with the execution of the instruction `wait` of the first waiting activation. The expression that appears in those instructions must be an identifier or `none` when the result of the corresponding call is not used or when there is no result. When the result of a call is used, the expression must be the name of the local variable that is assigned the result.

Instantiation of objects The creation of a new object is done by a bottom-up walk in the class hierarchy starting at the class of which the new node is an instance. Then, the constructor is treated like a standard method call except that there is no `wait` instruction that is inserted the creation instruction is removed from the instruction list.

2.3 Accessibility of Objects

Definition 1. *Accessibility*
For a given interpreter state with a call-stack S, the accessibility of an object o is the union of:

- *the set of objects transitively reachable from o (excluding o itself) through any reference path, and*
- *the set of objects that are referred by some local variable of an activation a such that there is an activation a' beneath a in S that points to o with its variable* `this`.

We write $\mathcal{A}(o)$ for the accessibility of o.

When an object is activated, all the method invocations on it are asynchronous. That is the reason why we define the accessibility of objects rather than the accessibility of method executions. Doing so, we capture the accessibilities of all method executions on an object.

We include into the set of operations performed by a method call all the operations that occur between the time the new activation a, corresponding to the method-call, is pushed onto the call-stack and the time it is popped. The object target of the method call corresponding to a is pointed to by the local variable `this` of a and an activation a' has to be included into the set of operations of a if it is above a into the call-stack. This explains the second part of the Definition 1.

2.4 Activable Objects

Now that we have defined accessibilities, we give a property on them which express that they have the same properties as sub-systems. This property is stated for a given interpreter state. For an object to be activable, it has actually to verify that property in all the states of its life-time (between the time it is created and the time it is no longer referenced or the program terminates).

Property 1. Activable
For a program P, in a given interpreter state S with a stack s and an object graph h, $Activable_S(o)$ iff

1. $\mathcal{A}(o)$ is a disjoint part of the object graph: for all object $o' \in \mathcal{A}(o)$, for all object o'' either:
   - $o'' \in \mathcal{A}(o)$ or
   - o belongs to all the paths from o'' to o'

   and
2. for all object $o' \in \mathcal{A}(o)$ for all activations a that refer to o' through a local variable `x`, there is an activation a' in the stack beneath a that refers to o through a local variable `this` and
3. no outgoing edge from o is a back-edge. A back-edge is an edge $u \rightarrow v$ such that u is a descendant of v in a depth-first walk of the object graph. We consider trees that are constructed starting from objects pointed to by local variables. Local variables are considered bottom-up in the call-stack.

This property is stated in three parts, the first two being related to the corresponding part of the definition of accessibilities. The first one ensures that any object in the accessibility

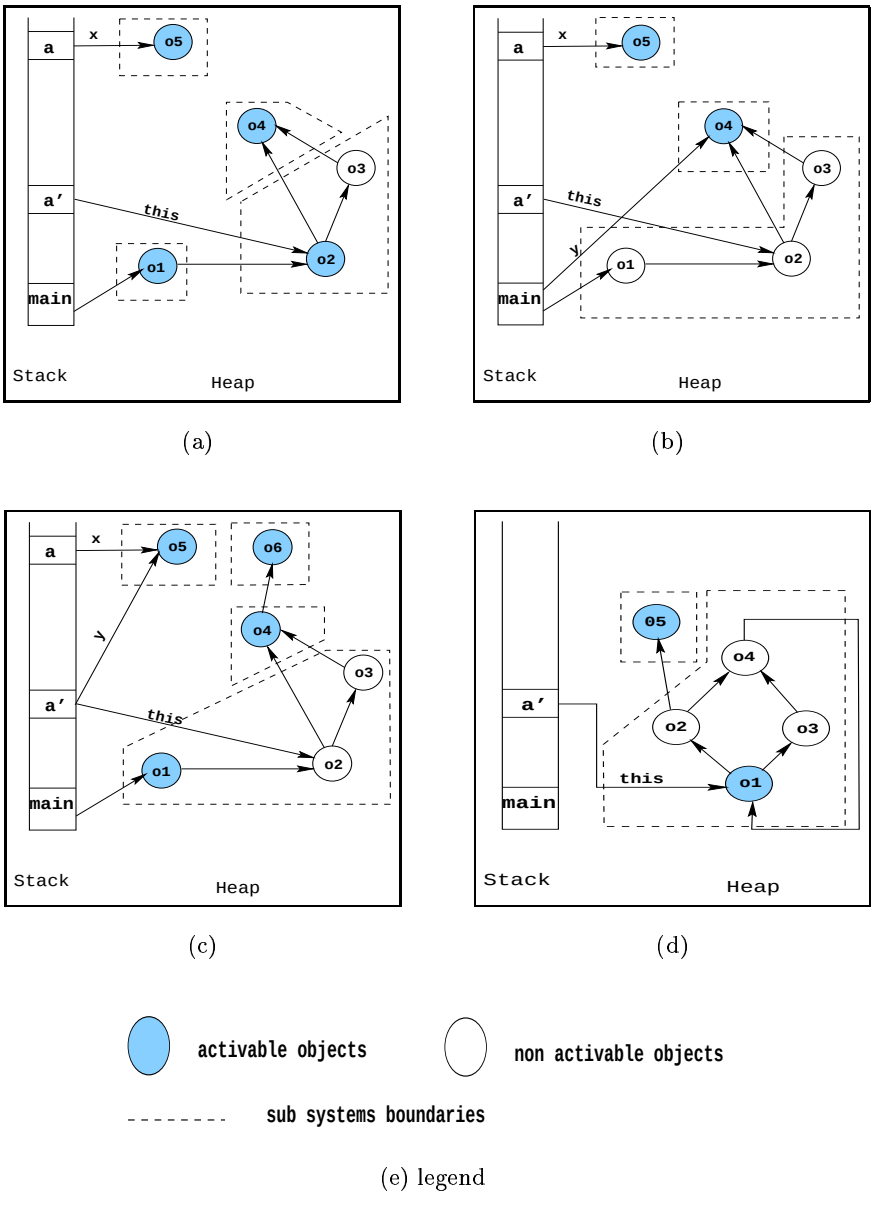

Fig. 5. Activable Objects

of an object o is only reachable through o. This is why the object o_3 on Figure 5(a) is not activable: the object o_4 is accessible from o_2, which is not in the accessibility of o_3, and it exists a path from o_2 to o_4 that does not contain o_3. This part guarantees that there will be no possible interference through the object graph.

The second part checks that no object in the accessibility of an object o is pointed to by a local variable from an activation that has not been initiated (transitively) by an activation that points to o with its variable `this`. The object o_2 in Figure 5(b) is no longer activable, as compared to its counter part of Figure 5(a), because the method `main` points to o_4. If the object o_2 were activated, the method `main` could trigger computations involving o_4 while the method call on o_2 would not be terminated and there would be a possible interference through o_4. Note that in this part, we do not take into account auxiliary variables introduced by the normalisation of programs.

The third part prevents us from building cycles between active objects. Cycles would allow interferences and would add the possibility of introducing deadlocks. All the objects represented in Figure 5(d) would be activable without the third part of the property; it is clearly incorrect to activate all those objects. We could have stated this part saying that an object should not be included into a cycle. However it would have prevented the object o_1 of Figure 5(d) to be activable.

The restriction we gave in the second part of the expression of the property 1 prevents method-call on active objects to receive arguments that are referred by a local variable or a parameter. If we consider the binary tree example, we can only activate them when the content of a node is a base type or has a copy semantics. There are several solutions to alleviate this restriction, which is often too strong. First, a simple extension is to consider cases where there are actually no calls (syntactically, no dot notations) on the parameters, for every statement of the routine. This case is actually very frequent, where parameters are just forwarded from one active object to another, with no reading or writing until some point. The binary tree falls in that category. Another property, that is more precise but a little harder to satisfy, is the fact that the parameters are not used on the path going from the candidate asynchronous call to a redefinition of their values, or the end of the routine. Also, note that argument passing of a newly created object (like `x.foo(new Bar())`) is always possible.

2.5 Topology of Sub-Systems

We have stated a condition on objects for detecting activable ones. We will now show that if all or part of the objects that satisfy that property are activated, then the topology of sub-systems of the parallel program obtained is a tree. We first give a definition of the connection graph between sub-systems (active object graph) then based on that definition and on the Property 1 we sketch a proof that those graphs are trees.

Definition 2. *Active Objects Graphs (AOG for short)*
The Active objects graph of a state $S = \langle s, h \rangle$ *is* $A_S = (N_A, E_A)$ *where:*

- $N_A = \{o \in h | Activable_S(o)\}$

– $\forall n, n' \in N_A, \langle n, n' \rangle \in E_A$ *iff* $n' \in \mathcal{A}(n)$ *and there is no* $n'' \in N_A$ *such that* $n'' \in \mathcal{A}(n)$ *and* $n' \in \mathcal{A}(n'')$.

For example, consider the object graph of Figure 5(c), the AOG associated with it is $(\{o_1, o_4, o_5, o_6\}, \{\langle o_1, o_4 \rangle, \langle o_1, o_5 \rangle \langle o_4, o_6 \rangle\})$. The second part of the definition ensures that we take the smallest relation between activable objects. In the previous example, there is no edge between o_1 and o_6.

We already now that AOG's can't include cycles. Let us show now that they are trees.

Proposition 1. *Given a state* $S = \langle s, h \rangle$ *any node of* A_S *has at most one predecessor.*

Proof. Let n be a node in N_{A_S} that has two predecessor n_1 and n_2. We will show that $n_1 \neq n_2$ leads to a contradiction.

From $n, n_1, n_2 \in N_{A_S}$ we get $Activable_S(n) \wedge Activable_S(n_1) \wedge Activable_S(n_2)$. From Definition 2 and $\langle n_1, n \rangle \in E_{A_S}$, we get
$n \in \mathcal{A}(n_1) \wedge \not\exists n', Activable_S(n') \wedge n' \in \mathcal{A}(n_1) \wedge n \in \mathcal{A}(n'$. This implies, together with $Activable_A(n_2)$, $n_2 \notin \mathcal{A}(n_1) \vee n_1 = n_2$. Similarly we get $n_1 \notin \mathcal{A}(n_2) \vee n_1 = n_2$. Let us assume that $n_1 \neq n_2$ then $n_1 \notin \mathcal{A}(n_2)$ and $n_2 \notin \mathcal{A}(n_1)$.
From $\langle n_2, n \rangle \in E_{A_S}$ we get $n \in \mathcal{A}(n_2)$ which implies that:

(i) there is a path in h from n_2 to n or
(ii) $\exists a \in s$ such that a refers to n through a local variable `x` and $\exists a_2 \in s$ beneath a such that a' refers to n_2 through its local variable `this` (from Definition 1).

Assume (i) hold. From $n_1 \notin \mathcal{A}(n_2)$, and Definition 1 we can get that there is no path in h from n_2 to n_1 so because of (i) there is a path p from n_2 to n such that n_1 does not belong to p which results in a contradiction with $Activable_S(n_1)$ because $n_2 \notin \mathcal{A}(n_1)$. From this, we get that (i) can not hold(simetrically, there is no path from n_a to n in h). So that (ii) must hold.

Assume (ii) hold. From $n_2 \notin \mathcal{A}(n_1)$, there is no activation a_2 in s referring to n_2 through some local variable such that there is an activation a_1 in s beneath a_2 that refers to n_1 through its local variable `this`. In particular there is no such a_2 that refers to n_2 through its local variable `this`. But (i) does not hold as previously seen so that we get $n \notin \mathcal{A}(n_2)$ which is in contradiction with $\langle n_2, n \rangle \in E_{A_S}$. So we have proved that if n has two predecessors n_1 and n_2 in N_{A_S} then $n_1 = n_2$. $\diamond$

AOG's are forests, and not only trees, because the method `main` may point to objects that are activable but with accessibilities that are disjoint: there is a tree for each such object in an AOG.

We have to justify now that the predicate $Activable_S$, when it is true for an object o on any state S where o has been created, permits to turn o to be active without modifying the results of the program. First we have to remark that if an object o is *activable* throughout an execution, then it never changes its parent in the AOG. Otherwise there would be an intermediate state where it is reachable from two other activable objects, which, as we have seen with Proposition 1 would actually prevents o from being activable. With that in mind,

we will show that the fact that AOG are forests permit to turn any object o that verifies $Activable_S$ on any state of a program run to be active.

The property $Activable_S$ being verified by an object o actually states that any object that is not in $accessibility(o)$ can only access elements of $accessibility(o)$ through o itself.

For that reason, any operation that is not targeted to object o is independent from any operations targeted to object o and can be safely done in any order relatively to operations on o. So the only thing we have to justify is that operations on o, which are assumed to be interdependent, are triggered, in the parallel program, in the same order as in the sequential one.

Within the model of execution of Java//, active objects have their own thread of execution and all passive objects that they directly access have their methods executed by that thread. Here, we say that an object o is *directly* accessed by an active object o_a if o is in the accessibility of o_a and there is no other activable object in $\mathcal{A}(o)$ such that o is in its accessibility. The set of passive objects thus defined is the sub-system of o_a.

If an active object receives its method calls in the same order as in the sequential program, then the order between operations performed by its thread is the same as the order they are performed within a sequential execution. From this, if o_a receives its method calls in the same order as it does in the sequential program and so do active objects directly accessed by o_a. To end up with this, we remark that the method `main` has its own thread of execution and that it triggers operations on objects of its sub-system in the same order as in the sequential program.

3 Object-oriented Static Analysis

We successively describe the two abstractions of the components of an interpreter state: call-stack and the graph of objects. These are not independent because the abstraction of graph of objects is done accordingly to a given state and the computation of the abstract transition relation may depend on the abstraction of graph of objects (dynamic dispatch).

3.1 Principles of abstract interpretation

Abstract interpretation [17] is a method for the design of static analysis. It is based on a connection between the semantics of programs and the analysis that allows to formally prove the correctness of the analysis. From operational semantics described as relation transitions on states, an analysis is designed in three steps as follows:

1. First, design a dynamic semantics (we call it the *standard semantics*) as we do in Section 2. We refer to the transition relation between states as τ.
2. Then, from the standard semantics define the *collecting* semantics to be the complete set of states reachable from initial states[1]. The collecting semantics (Φ) can be formally expressed as the least fixpoint of an operator that accumulates states using τ. The lattice used is the powerset of *State* ordered by set inclusion.

[1] Initial states must be specified so that all possible inputs are taken into account.

3. Last, design abstractions for domains used into the standard semantics. These abstractions must form a complete lattice and be connected to the collecting semantics by a pair (α, γ) of functions called a *galois connection.*

In the remainder of this section, we design a finite set of abstract states (the domain $State_a$) and a relation τ_a that approximate the relation τ. The result of the analysis is the least fixpoint of an operator build upon $\mathcal{P}(State_a)$. The correctness is ensured by the property of galois connections:

$$\mathcal{P}(State) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(State_a)$$

is a galois connection iff:

$$\forall s \in \mathcal{P}(State), \forall s_a \in \mathcal{P}(State_a), \alpha(s) \subseteq s_a \Leftrightarrow s \subseteq \gamma(s_a)$$

The relation τ_a must allow the computation of a fixpoint $\Phi_a \in \mathcal{P}(State_a)$ such that $\alpha(\Phi) \subseteq \Phi_a$ from what we can conclude $\Phi \subseteq \gamma(\Phi_a)$. This last result allows to prove that sufficient conditions hold on programs.

In this section, we describe the domains upon which are built our analyses. Then, in a subsequent section, we design tests on abstract domains that approximate the Property 1 in the sense described above.

3.2 Abstraction of Call-stacks

Because of recursive methods, call-stacks may be unbounded in size. To obtain a computable abstract semantics, we partition the set of states of the interpreter into a finite set of program points.

Activations are identified by the name of their corresponding method, the class where that method has been defined and the list of instructions. Because there is a finite number of methods in a program and method size is finite, we obtain a finite number of activations. They are represented by elements of the domain $P = inst \times Type \times Id$
where $Type$ is the set of class of the program analysed.

To approximate the relation caller/callee, we attach sets of return points to each element of P. The set of return points of an activation a is the set of method invocation sites (elements of P) that generated activations identical to a. It is used to compute the abstract transition from `return` instructions. The set of return points can be seen as edges between activations. Thus, we approximate stacks by graphs of abstract activations. Because the domain P is finite, such graphs are in finite numbers. An abstract stack is then an element of the domain

$$Stack_a = \mathcal{P}(P \times \mathcal{P}(P))$$

where the second component of elements are the sets of return points attached to abstract activation.

To make things clear, let us give the expression of the function that maps concrete stacks to their abstract counterpart. Given a stack $s \in Stack$, we note $\langle inst_i, \tau_i, \mu_i, \rho_i \rangle$ the activation at level $0 \leq i \leq |s|$ in s,where the level 0 is the bottom of the stack. The function $\hat{\alpha}_{stack} : Stack \rightarrow Stack_a$ is defined by:

$$\hat{\alpha}(s) = \bigcup_{1 \leq i \leq |s|} \langle inst_i, \tau_i, \mu_i, R_{\tau_i,\mu_i} \rangle$$

where:

$$\forall 1 \leq i \leq |s|, R_{\tau_i,\mu_i} = \{\langle inst_j, \tau_j, \mu_j \rangle | \tau_{j+1} = \tau_i \wedge \mu_{j+1} = \mu_i\}$$

The definition of R_{τ_i,μ_i} builds the set of call sites where abstract activations identified by τ_i and μ_i are created.

The reverse mapping is the set of paths starting from points corresponding to the method `main` and ending with a node that is not a waiting activation.

It is clear that he number of nodes of such a graph is bounded and so would be the number of such graphs. However, this method would be quite inefficient: the size of the result of the analysis might grow to be exponential in the number of methods of the analyzed program. Even if this is not likely to arise in practice, the size of the result would be quite large in any case. To overcome this problem, we actually use a single graph to represent all the call-stacks that may occur during any execution of the analysed program. It is the union of all the abstractions for call-stacks obtained by the previous method where the sets of return points of identical activations are merged. Flow-sensitivity[2] is still achieved by attaching dataflow information to those program points that correspond to non-waiting activations. This leads to a result that is linearly bounded by the size of the program. Such an abstraction for program points is close to the notion of control-flow graph [1]: it is isomorphic to the graph of the abstract transition relation between program points.

3.3 Abstraction of Object Graphs

The size of heaps may be unbounded because classes may describe recursive structures. A common way to finitely abstract sets of heaps is to map concrete references to a finite set of abstract references. Sagiv et al. [37] proposed to use sets of variables to name objects: an object is mapped to an abstract object whose name contains all the variables that point to it. Then, the number of variables is finite because they present an intra-procedural analysis. In our case, it may be unbounded because of recursive methods. We limit the number of variables using a single representative for waiting[3] recursive activation. Local variables are depicted by the name of their corresponding methods, the name of the class where that method has been defined and the name of the variable (to distinguish between variables of different activations that have the same name). We depict variables from waiting activations

[2] An analysis is said *flow-sensitive* if it computes information that is program point specific.

[3] waiting abstract activations are representatives of concrete activations that are not on top of the call-stack.

by an over-lined notation and we add a special symbol ($\Box$) to depict the place-holder for values. The naming scheme for waiting activations guarantees that there is no confusion between the waiting activations and the current one in the case of recursive methods. As such, it is possible to treat recursive activations in the same way we do non-recursive ones. The domains to describe variables are:

$$\begin{array}{ll} Name & = Ident \cup (O_i)_{i \in \mathbb{N}} \\ Var & = (Type \times Ident \times Name) \cup \{\Box\} \\ \overline{Var} & = Type \times Ident \times Name \end{array}$$

The set we use as abstract references is a combination of classes, allocation sites and sets of variables:

$$Ref_a = Type \times P \times \mathcal{P}(Var \cup \overline{Var})$$

The second component of abstract reference is the program point where the abstract object having that reference has been instantiated.

Abstract heaps are described using the domains

$$\begin{array}{ll} Val_a & = \mathcal{P}(Ref_a) \cup \{int, bool\} \\ Obj_a & = Ref_a \times Type \times (Id \rightarrow Val_a) \times \{\text{true}, \text{false}\} \\ Objs_a & = \mathcal{P}(Obj_a) \end{array}$$

We drop information about scalar types (integers and booleans) because we do not need them to represent object graphs. The domains Int and $Bool$ are respectively abstracted to singletons $\{int\}$ and $\{bool\}$.

Abstract values representing references are actually sets of abstract references. Consider for instance that the function α^S_{Ref} maps concrete references to the class of the corresponding objects (a single abstract object represents all the instances of a given class). In a given object graph, distinct instances of a class, say c, may point to objects of distinct classes $(c_1, \ldots, c_n)$ through a given instance variable, say x. The abstraction of those objects of class c will all be mapped to the same abstract object (with the abstract reference c) and the instance variable x will have to point to abstract objects having different abstract references $(c_1, \ldots, c_n)$.

Abstraction of objects mimics the form of concrete ones. It includes the class of the objects (because we partition the set of concrete objects using classes, an abstract object represents instances of a single class); it also includes a mapping from identifiers to abstract values that is an abstraction of the attribute list of objects and a binary attribute that reports the fact that an abstract object represents shared concrete objects or not. A concrete object is shared if at least two distinct attributes (from distinct objects or not) point to it. This boolean is essentially what permits us to check whether *Activable* may hold or not on abstract objects. In addition it is also important for the abstract interpretation of operations on objects.

The powerset forms a complete lattice, where order is the set inclusion and join (respectively meet) operation is union (respectively intersection). The set of abstract object graphs

forms a complete lattice where the join operations perform the union of two object graphs, merging identical objects together with formal expression of abstraction functions.
Finally our analysis computes a set of elements from the domain:

$$State_a = P \times \mathcal{P}(P) \times \{\text{true}, \text{false}\} \times Env_a \times Object_a \times Val_a$$

where the first three components are the complete abstraction of stacks (program point, return points and a boolean attribute that is true when the activation may be a recursive one). In addition, abstract states have attached:

- an element of Env_a that is a monolithic[4] environment that records the mapping from any local variable in any call-stack in which the current abstract program point can be the top,
- an abstract graph of objects, and
- an abstract value for representing the place-holder for values.

Because we attach dataflow information (abstract environments and abstraction of graph of objects) to individual program points, the result reflects the set of stacks of the collecting semantics by a single graph (induced by the return points attached to abstract states) at the same time it achieves flow sensitivity.

3.4 Abstract Transition Relation

We briefly describe the transformations of object graphs corresponding to the different transitions.

Intra-procedural transitions Let us recall the different operations that we have to interpret on the abstractions for object graphs:

1. nullification of a local variable: operation of the form `x = null;`,
2. nullification of an object attribute: operation of the form `x.s = null`,
3. local variable assignments: operations of the form `x=y`,
4. assignment into an object attribute: operation of the form `x.s = y` and
5. assignment of an object attribute to a local variable: operation of the form `x = y.s`.

For interpreting the operations of type 1, it suffices to remove `x` from any abstract object name where it appears and to compact the result (merge objects with identical names). The environment is modified so that the variable `x` now has the value {`null`}.

For operations of type 2, we replace the current value of the attribute `s` of any abstract object pointed to by `x` by the value {`null`}. The sharing attribute of abstract objects is

[4] In the current implementation, only local environments are propagated. The environments when returning to a waiting activation can be built back from the abstract object graph.

updated so that nodes that are *singular*[5] nodes and for which the number of referring abstract objects is now one are marked as *non-shared*.

Operations of type 3 are interpreted by adding the variable `x` into the name of any abstract object pointed to by the variable `y`. Then the value for `x` (respectively for `y`) in the environment is replaced by the set of nodes that have `x` (respectively `y`) in their name.

Operations of type 4 are interpreted as the previous ones: we set the value of the attribute `s` of any abstract object having `x` in its name to be the set of abstract references that contain `y`.

Operations of type 5 are the most difficult to interpret. Consider the set $\mathcal{S}$ of abstract objects pointed to by `y.s`. These nodes are *unfolded* by adding the variable `x` into the names of copies of objects in $\mathcal{S}$. Then these new nodes are connected to the objects that previously pointed to them. Using the boolean attribute that records the sharing, it is possible to restrict the number of edges added: any non-shared abstract object is only added edges labeled with `s` from abstract objects pointed to by `x`.

The power of the naming scheme that uses variables comes from the fact that nullification is possible on attributes of abstract objects. Operations are interpreted on nodes that have variables into their name. This implies that these nodes reflect each a **single** node in each concrete state that is mapped to the current abstract one so that it is possible to remove the value in attributes of objects. Otherwise we could only merge the new value with the old one so that the result safely reflects all the objects that are mapped to the modified one.

Inter-procedural transitions

On a call site, the abstract transition rule generates as many states as there are abstract references in the evaluated target object (actually one per class represented in the evaluated value). For each of those, it get method bodies using the same function as in the concrete semantics (only classes are needed to do the method lookup). Then, each parameter is assigned its value (which has been evaluated prior to the call). For parameter that are of type references, the assignment is simply interpreted like an intra-procedural operation of type 1. A temporary name is used to name the called method, which is replaced by the actual method name after the abstract object graph has been folded to limit the number of nodes because of potential recursive activations.

When returning from an activation, all the local variables of the terminating activation are nullified by transforming the object graph as it is done for operations of the kind `x = null`.

When returning to a recursive activation, we have to unfold the abstract object graph so that variable of the new current activation point to singular node (nodes that have only over-lined variables in their name represent multiple concrete object).

[5] Nodes are singular if they contain at least one variable from a non-recursive activation in their name. Non singular nodes are referred to as *summary* nodes.

```
class ListInsert {
   public static void main(String[] args) {
      Node head,n;
      int i,j;
      head = new Node();
      i = 2;
      while (i < 5) {
         n = new Node();
         n.next = head;
         head = n;
         i = i + 1;
      }
      n = head.next;   //S1
      aux = n.next;    //S2
      head.next = aux; //S3
      n.next = head;   //S4
      head = n;        //S5
}}
class Node {
   Node next;
}
```

Fig. 6. A simple linked list construction

Abstract interpretation of all the forms of assignments is illustrated in Figure 5 where the object graphs occurring respectively before statement `S1` of the program shown in Figure 5 and after statements `S2,S3,S4`.

Abstract Garbage Collection

To get a more accurate representation of heaps, we added to the transformations defined in [37] an abstract garbage collection operation. This operation permits to obtain precise results when interpreting the *remove* operation on linked lists. When this is not done, removed nodes still point to the node that replaced them. As such, nodes that replaced removed ones are marked as *shared*. As we will see in the next section, the sharing attribute is crucial in the detectection of activable objects on abstract object graphs.

4 Applications and examples

This section first explains how to exploit the result of the static analysis. Then, we present an example of object activation: a Sieve program used to generate prime numbers (see Figure 4.1). We use this example to illustrate how we detect activable objects on abstract grahs.

4.1 Example: the program Sieve

We present here another example of code that can be parallelised. The program Sieve computes successive prime numbers. It builds a linked list of instances of the class `Prime` that

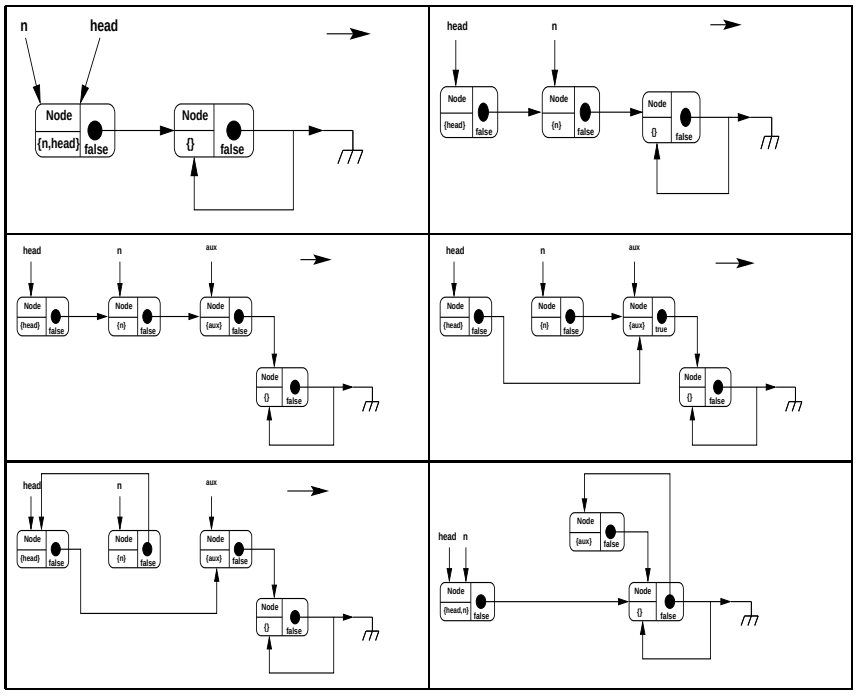

Fig. 7. Inversion of two elements of a linked list

represents prime numbers. The main method iterates on natural numbers up to the maximum desired and propagate them into the list. Non prime numbers are detected by division by the prime numbers already into the list and when a divisor is found for a number, that number is rejected. New prime numbers are added at the end of the list.

On this program, our analysis report that instances of the class `Prime` are never shared (the shape is a linked list). Thus, we suceed to prove that accessibilities of those objects are self-contained. Any object instantiated within the method `sieve` is pointed to only by auxiliary variables so that the second part of the property 1 is verified as well. The object pointed to by the variable `p` is not affected by this condition because we look for local variables in the name of nodes that are in the accessibility of the node we want to activate. Then it is possible to activate any instance of the class `Prime`. The line labeled with (*1) can be replaced with:

```
p = (Prime) Javall.newActive ("Prime", 2, null);
```

and the line (*2) with:

```
aux2 = (Prime) Javall.newActive ("Prime", i, null);
```

```
class Sieve {
   public
   static
   void main(String[] argv) {
      Prime p;
      p = null;
      p = new Prime(2); //(*1)
      int i;
      i = 3;
      while(i <= 20) {
         p.sieve(i);
         i = i + 1;
}}}
class Prime {
   int number;
   Prime next;
   Prime(int number) {
      this.number = number;
      this.next = null;
}}
```

```
   void sieve(int i) {
      Prime aux1;
      Prime aux2;
      if (i % this.number != 0)
          if(this.next != null) {
             aux1 = this.next;
             aux1.sieve(i);
          } else {
             aux2 = new Prime(i); //(*2)
             this.next = aux;
          }
   }
}
```

Fig. 8. Program Sieve (sequential)

This example exhibits, as the binary tree example, pipeline parallelism. When a instance of `Prime` has propagated a number to its successor, it is available for receiving new numbers. The transformation needed to produce the parallel version of the program is the same as those applied to the binary tree example: a subclass of `Prime` that implements the interface `Active` is writen. Instantiation sites are replaced by calls to the routine `newActive` of the Java// library.

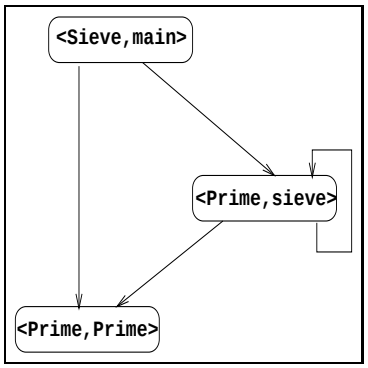

Fig. 9. Call-graph of the program Sieve

4.2 Approximate test for activable objects

Test for part 2 of the Property 1 That part of the property uses information on the stacks. To approximate it, we use an abstraction that can be computed from the sets of return points attached to abstract activations: the call-graph.

Definition of call-graphs Methods are uniquely identified by pairs of identifiers that are the name of the method and the class in which it has been defined: $Mid = Type \times Id$.

The call-graph of a program P is a graph where nodes are elements of Mid. There are edges between nodes m_1 and m_2 if m_1 calls m_2 during some execution of P (It is straightforward to obtain it from abstract activations and sets of return points). Recursive calls correspond to cycles in this graph. The Figure 4.1 shows the call-graph corresponding to the program Sieve shown in Figure 4.1. From now on, we refer to the call-graph as CG.

Consider the graph $CG^\star = (N^\star, E^\star)$, that is, CG reduced to its strongly connected components. We define the order $(Mid, \preceq)$ by $n \preceq n'$ iff n does not depict a recursive method and $n = n'$ or n and n' are not in the same strongly connected component and there is a path in $CG^\star$ from N_n to $N_{n'}$ (these nodes are respectively the strongly connected components of n and n')[6]. That order can be interpreted as $n \preceq n'$ iff any activation mapped to n always appears *under* any activations mapped to n' in call-stacks.

Overview of the test To satisfy the second part of the property 1, a node must be pointed to by no local variable or only by local variables from activations that have been triggered by an activation called on the object under interest.

The corresponding approximate test on abstract objects examines the set of variables (from the domain Var) in the name of a node and compares their activations using the order $\preceq$. An abstract object o with name $\langle \tau, p, vars \rangle$ satisfies the property if:

[6] Because the reduction of an arbitrary graph to its strongly connected components yields a DAG, the relation $\preceq$ satisfies axioms of partial orders.

1. for any $\overline{\langle \tau, \mu, x \rangle} \in vars, x \neq this$, there is a variable $\overline{\langle \tau', \mu', this \rangle} \in vars$ such that $\langle \tau', \mu' \rangle \preceq \langle \tau, \mu \rangle$.
2. for any $\langle \tau, \mu, x \rangle \in vars, x \neq this$ there is a variable $\overline{\langle \tau', \mu', this \rangle} \in vars$ or there is a variable $\langle \tau, \mu, this \rangle \in vars$. In that case, any overlined variable is necessarily from an activation that is under the current one in the call-stack: overlined variables are variables from waiting activations.

Testing if an abstract object has a self-contained accessibility

Approximation of Accessibilities We define here the accessibility of an abstract object. Given a call graph CG and an abstract object-graph H, the accessibility of a node $o = \langle \tau, vars \rangle \in N_H$ (denoted $\mathcal{A}_a(o)$) is the set of nodes reachable from o in H plus the set of nodes that have $\langle \tau, \mu, \_ \rangle$ or $\overline{\langle \tau, \mu, \_ \rangle}$ in their name such that o have $\langle \tau_o, \mu_o, this \rangle$ in its identifier and $\langle \tau_o, \mu_o \rangle \preceq \langle \tau, \mu \rangle$.

Compatibility of nodes An abstract object graph records properties of several (possibly infinitely many) concrete object graphs. It is still possible for some nodes to verify if their respective concretization may appear in a same concrete object graph. In a concrete state, a variable points to a single node, so that if there are different nodes of an abstract object graph that have a given local variable in their name then they can be representative of nodes of a same concrete state only if the considered variable may represent several concrete variables (variable in $\overline{Var}$). So we define the predicate *compat* that is verified by two nodes if they *may* be a representation of concrete nodes appearing in the same state.
Let $n_1 = \langle \langle \tau_1, vars_1 \rangle, \_, \_, \_ \rangle$ and $n_2 = \langle \langle \tau_2, vars_2 \rangle, \_, \_, \_ \rangle$, the predicate *compat* is defined as follows:

$$compat(n_1, n_2) = vars_1 = vars_2 \vee \forall v \in vars_1 \cap vars_2, v \in \overline{Var}$$

The relation induced by this predicate is symmetric but it is not transitive: $\{x\}$ and $\{y\}$ are compatible and so are $\{y\}$ and $\{x, z\}$ but $\{x\}$ and $\{x, z\}$ are not. To extend *compat* on sets of nodes, we impose that all pairs of nodes of a set are compatible.

We define the predicate $path - compat$ as follows:
Three nodes n, n' and m verify $path - compat$ if

1. $compat_a(n, n', m)$ and
2. there is a path $n(n_i)_{i \leq k} n'$ in G_a such that $compatible_a(\{n_i | 1 \leq i \leq k\} \cup \{n, n'\})$

Abstraction of $\notin$ Given an abstract object graph H and $S \subseteq N_H$ a subset of objects in H, we define the predicate $\notin_a$ so that it is a safe approximation of $\notin$ on concrete object graphs. Given a set of abstract objects S and an abstract object $o \in S$, if o is not singular then S may represent some concrete object set S_c and o a concrete object that is not in S_c. So that it is not safe to use $\notin$ on abstract objects graphs to approximate $\notin$ on concrete object graphs.

Let S be a subset of N_{G_a} and $m \in N_{G_a}$ then

$$m \notin_a S \text{ if } m \notin S \vee m \in S \wedge \neg singular(m)$$

A node is said to be singular if it reflects a single concrete object in each concrete state of the concretization of the current abstract one. A node is singular if it has a variable, from an activation that is not recursive, in its name.

Approximate property on accessibilities of abstract objects We now have all the elements to express a safe approximation of the property $Activable$. We denote that abstract predicate $Activable_a$.

Let H be an abstract object graph and o an abstract object in H. Then, o will be rejected ($notActivable_a(o)$) if:

$$\exists m \in \mathcal{A}_a(o), is\_shared(m) \wedge \exists m' \in H, m'[s] = m, \\ path - compat(o, m, m') \wedge m' \notin_a \mathcal{A}_a(o).$$

Detection of cycles on abstract object graphs We sketch here an algorithm that detects possible cycles in abstract object graphs. It is based on a depth-first search that checks, for each new visited node, whether it is already on the stack. If a node is accessed from the top of the stack, and it is already on the stack, the edge used to access it is clearly a back-edge.

The algorithm provides only an approximate answer (within reason, the abstract object graph is only an approximation of all the object graphs that may occur at the program point under interest): it gives a super-set of back-edges. As such, it is a safe approximation of the sets of back edges on concrete objects graphs.

The nodes have to be considered in the order induced by $\preceq$: nodes pointed to by variable from an activation $n \preceq n'$ will be considered prior to those of n'. The algorithm performs a depth-first walk on the graph and pushes onto its stack only nodes that are compatible with all the nodes already into the stack. So, only *valid* paths in an object graph are considered.

If a node is selected and there is a node into the stack that may reflect the same concrete object, then the selected edge makes up a cycle and the node on top of the stack can not be activated.
A node may make up a cycle if:

- it is already into the stack and it is the bottom of the stack or
- it is not the bottom of the stack and it is shared.

We make the distinction between nodes that are on the bottom of the stack and others. A node that is not on the bottom of the stack has been accessed through an instance variable s. If it is accessible from the top of the stack (a distinct node), then it must be accessed through an instance variable that is not s; otherwise, there would be cycles in the stack. If the considered abstract object is not shared, then the newly accessed object in that context is the representative of concrete objects that are distinct from those represented by its

counterpart that is already into the stack. This cannot be said for the node that is on the bottom of the stack because it has not been accessed through any instance variable.

Finally, activable objects are detected on abstract graphs

5 Related Work

5.1 Static analysis of object-oriented programs

One of the major problem when analyzing object-oriented programs is that the flow of control cannot be determined statically because of dynamic dispatch. We have seen that the interpretation of method calls involved method lookup for each class determined by the abstract evaluation of the target object of the call. And because we partition the set of concrete objects using classes, we restrict the number of actual classes for target objects of method calls. This problem, known as *class analysis, concrete type inference* or *control flow analysis* have been investigated by many authors. Class analysis is motivated by compiler optimizations that are now traditional in the field of object-oriented languages (removal of dynamic dispatch, method in-lining), but solutions to this problem are needed to get a good solution to any interprocedural problem. Different approaches have been investigated as described.

Dean, Grove and Chambers [19] proposed an analysis called *static class hierarchy analysis*. This analysis tends to use as much as possible static information that can be drawn from the class hierarchy. After a fast intra-procedural class analysis(designed as a dataflow analysis), the class hierarchy is analysed to determine, for each set of classes infered for lexical method calls, those sets where the given method is never overriden. For these method calls, it is possible to remove dynamic dispatch. In the case of statically typed languages, as Java, typechecking can be used in place of dataflow analysis.

Pleviak and Chien [35], and Defouw, Grove and Chambers [20] present analysis based on constraint systems. These analyses compute a fixpoint by iterating on structures called *data-flow graphs* where nodes represent variables, object creation sites and method-calls, and edges assignment between variables, assignment of a newly created object and assignment of actual to formal parameters on method-calls. Edges are interpreted as constraints on sets of class that can be formally expressed by set inclusion: an edge $u \rightarrow v$ (from a statement `v:=u` for instance) corresponds to the constraint $[v] \subseteq [v]$. The solution of a constraint system is obtained by a fixpoint computation.

Grove, Defouw, Dean and Chambers [24] present an analysis that computes an approximation of call-graphs. The reasons why call-graphs determination needs a static analysis are the same as those that motivate class analysis. Yet the result may be used to support subsequent interprocedural analysis. This is a constraint based analysis as the one presented in [20]. All these analysis have some sort of abstract representation of heaps because they infer set of classes for instance variables as well as for local variables and formal parameters. But they do not have the possibility to nullify instance variables because a single abstract representative of instance variable represents, in general, instance variable of several concrete instances.

Chatterjee et al. [15,16] present an analysis for programs that have features of both C++ and Java excluding threads and *finalizers* methods for Java. The analysis is essentially *points-to analysis* as introduced by Emami et al. [31]. It aims at detecting what objects variables point to at a given program point. They limit the number of abstract objects using classes and instantiation sites to form a limited number of representative of concrete objects. They adopt a two-level approach to interprocedural analysis (as presented in [1]) that computes, for each method, a function that summarizes the effect of the method on any context. This approach allows them to maintain only part of the program in the main memory at a given time, because the top-level analysis is a bottom-up traversal of a DAG reduction of the call-graph. They can perform *destructive-updates* in some cases because they differentiate initial values for formal parameters on method calls from any object allocated at any subsequent site; but they can not do systematic *destructive updates* and the heap representation they obtain does not reflect sharing properties. Two-level interprocedural analysis is known to be more space and time efficient than the other approach that consists in building a super control flow graph, connecting call sites with entry nodes of methods and exit nodes with return sites. This is because they can summarize the effect of methods independently from any context and then apply the summary functions to a given context to obtain a context-sensitive analysis. Thus a given method is analyzed once for all. It is unclear that the object graph representation we use can lend itself to the two-level approach for interprocedural analysis as noted by Ross and Sagiv in [36].

Besides the work on class analysis, Almeida presented in [2] a type-system that relies on an abstract interpretation of object-oriented programs as a checking system. Classical type systems for object oriented programs are extended to incorporate the possibility for objects of sharing states as a first class property (classes of these objects are called *balloon types*). The abstract interpretation approximates an invariant that ensures for balloon types objects that they are self-contained. This work seems quite close to ours as far as analysis is concerned; however, we seem to have definitely different motivations and goals. Although the invariant that is enforced for balloon types is close to our notion of sub-systems, an important difference is that our sub-systems allow a certain form of sharing that is critical for distribution, reuse, and is more flexible: passive object can reference an active root object of another sub-system.

5.2 Shape Analysis

Shape analysis is a generic name for analysis that aims at statically infering properties on dynamically allocated structures. Many authors before Sagiv, Reps and Wilhelm investigated shape analysis using a finite naming scheme for nodes [29,34,25,14]. Some of them use *k-limiting* schemes (nodes that are accessed from variables by some paths of length exceeding a parameter k are clustered to a single summary node). Chase, Wegman and Zadeck [14] use allocation sites to partition the set of concrete nodes, but they can't do systematic destructive-updates and their methods can't infer such properties as the method presented in [38].

Some authors investigated *storeless* approaches where only aliased access path are represented. Hendren and Nicolau [27] use matrix of access paths represented using a limited form of regular expression. According to Sagiv, Reps and Wilhelm, this method can be as powerful as theirs in some cases, but it can't deal with cyclic structures and as such, can't be used for general programs. Ghiya and Hendren [22] present a method which infer whether structures pointed to by variables are tree-like, dag-like or general graphs. It provides information about the structure accessible from pointer of a given procedure but it is not easy to obtain global information on the objects graph with it. In contrast, it provides information about the structure accessible from pointer of a given procedure.

We have enhanced the analysis of Sagiv et al. [37,38] in several ways. To our knowledge, there is no description of an interprocedural extension of their analysis that has been published. Beside this, we include into our analysis a garbage collection on abstract heaps. It is able to detect garbage in the case of the removal of a node in a linked-list. This point is particularly important because it permits the analysis to detect that a list is still a list after an arbitrary number of `remove` operations.

6 Conclusion

In this article, we investigated the use of static analysis for building distributed and parallel programs in Java.

We propose to analyse a sequential program, and to exploit the results of this analysis with the intervention of the (end-)user, in order to provide a parallel version of the original program which has two properties: (1) reuse with as few modifications as possible, (2) guaranteed preservation of sequential semantics. The resulting programs use primitives of Java//, a Java library for distributed, concurrent and parallel computing. Our analyses are based on abstract interpretation techniques and are powerful enough to statically determine a suboptimal list of objects that can be turned into active objects, and guide a transformation of the given sequential program into its parallelized version (as shown in the Sieve example).

We have described our analysis within Centaur [9]; this provides us with an executable version of the analysis which actually runs on the case studies presented (binary tree, linked lists, sieve).

Our work differs from that of Sagiv, Reps and Wilhelm [37,38] in that we are in an object-oriented setting, with the goal of parallelization and distribution. In addition, we have proposed an interprocedural analysis and a garbage collection extension.

Future work includes refinements of this analysis in order to detect more complex cases, and derivation of interactive tools in order to help the user to make decision on how to distribute his program. We also want to explore the application of this analysis in optimization of Java compilation.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers : principles technics and tools.* Addison-Wesley, 1986.
2. P. S. Almeida. Balloon types: controlling sharing of states in data types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS, pages 32–59. Springer Verlag, 1997.
3. P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proc. ECOOP '87*, LNCS 276, pages 234–242, Paris, France, June 1987.
4. I. Attali, D. Caromel, and S. O. Ehmety. About the automatic continuations in the Eiffel// Model. In *Proc. of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, Las Vegas, USA, Juillet 1998.
5. I. Attali, D. Caromel, and S. O. Ehmety. Formal Properties of the Eiffel// Model. In *Object Based Parallel and Distributed Computing.* Hermes, France, 1998. To appear.
6. I. Attali, D. Caromel, and S. Lippi. From a specification to an equivalence proof in object-oriented parallelism. 1998. submitted to publication.
7. I. Attali, D. Caromel, and M. Russo. A formal executable semantics for java. In *Proceedings of the Workshop on Formal Underpinning of Java*, 1998.
8. J. K. Bennett. The design and implementation of Distributed Smalltalk. In *Proc. OOPSLA '87, ACM SIGPLAN Notices 22 (12)*, pages 318–330, December 1987.
9. P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahna nd B. Lang, and V. Pascual. Centaur: the System. In *SIGSOFT'88 Third Annual Symposium on Software Development Environments*, Boston, 1988. ftp://babar.inria.fr/pub/centaur/papers/sde3.ps.
10. D. Caromel. Concurrency and reusability: From sequential to parallel. *Journal of Object-Oriented Programming, 3(3)*, 1990.
11. D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.
12. D. Caromel, F. Belloncle, and Y. Roudier. The C++// Language. In *Parallel Programming using C++*, pages 257–296. MIT Press, 1996. ISBN 0-262-73118-5.
13. D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, 1998. To appear.
14. D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointer and structures. In *Conference on Programming Languages Design and Implementation*, volume 25(6), pages 296–310. ACM, june 1990.
15. Ramkrishna Chatterjee, Barbara G. Ryder, and William A Landi. Relevant context inference. Technical Report DCS-TR-360, Department of Computer Science, Rutgers University, August 1998.
16. Ramkrishna Chatterjee, Barbara G. Ryder, and William A Landi. Relevant context inference. In *Proceedings of 26th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1999.
17. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM press, 1977.
18. P. Cousot and Radhia Cousot. Inductive definitions, semantics, and abstract interpretation. In *Proceedings of the 19th ACM Symp on Principles of Programming Languages*, pages 83–94. ACM press, 1992.

19. J. Dean, D. Grove, and C. Chambers. Optimization of object oriented-programs using static class hierarchy analysis. In *Proceedings of ECOOP'95*, Aarhus, Denmark, August 1995. Springer Verlag.
20. G. Defouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Proc. of Principles of Programming Languages*, january 1998.
21. P. Fradet, R. Gaugne, and D. Le Métayer. Static detection of pointer errors: an axiomatisation and a checking algorithm. In *Proc. European Symposium on Programming, ESOP'96*, volume 1058 of *LNCS*, pages 125–140, Linköping, Sweden, April 1996. Springer-Verlag.
22. Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, St. Petersburg Beach, Florida, 21–24 January 1996.
23. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
24. D. Grove, G. Defouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *OOPSLA'97 Conference*, 1997.
25. V. A. Guarna. A technique for analysing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 212–220, 1988.
26. L. J. Hendren, M. Emami, R. Ghyia, and C. Verbrugge. A practical context-sensitive interprocedural analysis framework for c compilers. ACAPS Technical Memo 72, School of Computer Science, McGill University, Montréal, Québec, 1993.
27. L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, 1990.
28. S. Hodges and C. B. Jones. Non-interferance properties of a concurrent object-based langu age: Proofs based on an operational semantics. In *Object Orientation with Parallelism and Persistance*. Kluwer Academic Publishers, 1996. ISBN 0-7923-9770-3.
29. J.R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 21–34, june 1988.
30. X. Liu and D. Walker. Confluence of processes and systems of objects. In *Proc. of Theory and Practice of Software Development (TAPSOFT'95) 6th International Joint Conference CAAP/FASE*, 1995.
31. L. J. Hendren M. Emami, R. Ghiya. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.
32. O. Nierstrasz. Active objects in hybrid. In *Proc. OOPSLA '87, ACM SIGPLAN Notices 22 (12)*, pages 243–253, 1987.
33. Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages p. 161–170. ACM Press, New York, 1998.
34. J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for parallel execution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes In Computer Science, pages 37–56. Springer Verlag, 1993 1993.
35. J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices*, 29(10):324–??, 1994.
36. J. Ross and M Sagiv. Building a bridge between pointer aliases and program dependences. In *Proceedings of European Symposium on Programming*, volume 1381 of *LNCS*, April 1998.

37. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape analysis problems in languages with destructive updating. In *Conference Record of the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM press, january 1996.
38. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.
39. C. F. Schaefer and N. G. N. Bundy. Static analysis of exception handling in ada. *Software Practice and Experience*, 23(10):1157–1174, October 1993.
40. D. A. Schmidt. Natural-semantics-based abstract interpretation. In A. Mycroft, editor, *Static Analysis Symposium*, number 983 in LNCS, pages 1–18. Springer Verlag, 1995.
41. D. A. Schmidt. Abstract interpretation of small-step semantics. In Springer Verlag, editor, *Proc. 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, number 1192 in LNCS, pages 76–99, June 1996.
42. M. Tokoro and K. Takashio. Toward languages and formal systems for distributed computing. In *Proc. of the ECOOP '93 Workshop on Object-Based Distributed Programming*, LNCS 791, pages 93–110, 1994.
43. T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proc. OOPSLA '88, ACM SIGPLAN Notices 23 (11)*, pages 306–315, November 1988.
44. Y. Yokote and M. Tokoro. The design and implementation of Concurrent Smalltalk. In *Proc. OOPSLA '86, ACM SIGPLAN Notices, 21 (11)*, pages 331–340, November 1986.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399